65103
March 29, 1971

STRUCTURE OF A MULTIPROCESSOR USING MICROPROGRAMMABLE BUILDING BLOCKS

BY

R. L. Davis

S. Zucker

Presented
at the
National Aerospace Electronics Conference 1971

Burroughs Corporation

Defense, Space and Special Systems Group

STRUCTURE OF A MULTIPROCESSOR USING MICROPROGRAMMABLE BUILDING BLOCKS

R. L. DAVIS and S. ZUCKER

Advanced Development, Burroughs Corporation, Paoli, Penna.

ABSTRACT

This paper describes a general purpose, microprogrammable, hardware building block called an Interpreter, an LSI-multiprocessing system in which the Interpreter is used, and a machine structure, implemented via firmware on the Interpreter.

The Interpreter consists of five types of functional modules, each partitioned for eventual implementation with LSI arrays of 450-750 gates and less than 126 signal pins. One of these functional units is the writable microprogram memory, whose contents define the function of the Interpreter. The flexibility of the Interpreter is typified by its present use as a device controller, a stand-alone emulator of other machines, and as a multiprocessor.

The Interpreter's main design concept allows functions formerly performed by software alone to be now performed by the emulating hardware. In this presentation an instruction set, which includes those software functions frequently and consistently used in operating systems, is emulated. The order code will allow for easy table and list manipulation or handling since much of scheduling and resource handling is confined to such operations.

The ability to write code independent of the data to be processed is provided by accessing information through descriptions. These descriptions can locate the requested information, describe its structure, impose controls on the use of the information and provide signals to the operating system for special functions.

INTRODUCTION

Presently, the trend in computer science is toward multiprocessing systems which can increase processing efficiency, reliability and throughput. The control of multiprocessing is more complex than that of conventional systems. Methods of development of these systems have in the past been limited by the hardware upon which they are implemented. With the advent of microprogrammed computers, firmware can now be developed to emulate a machine structure which can handle control structures more efficiently.

The concepts of microprogramming [1, 2, 3, 4], multiprocessing [5, 6, 7, 8] and avionics computers [9, 10] have been well covered. The purpose of this paper is to present, in Part I, a brief description of the Interpreter (a microprogrammable, hardware building block) and the Switch Interlock (a building block used to interconnect a configuration of Interpreters as a multiprocessor), and, in Part II, the description of an approach to writing an operating system for such a multiprocessor.

PART I. INTERPRETER BASED SYSTEMS

Interpreter-based systems emphasize two basic concepts: building block structure and soft machine architecture through microprogramming. The microprogrammable buiilding block structure was developed primarily due to the constraint of a design predicated upon eventual implementation of each functional unit with LSI. This constraint forced simplicity, modularity and versatility into the system. This gives a design usable in a wide range of applications, which is the only way to achieve a sufficient volume of parts to make LSI economically feasible. This resulted in a machine that is:

1. microprogrammable, to avoid irregular control logic and to provide the versatility required,
2. modular in 8-bits in the functional unit that performs the logic, arithmetic, and shifting in order to match the word length to the problem (important for emulation), and
3. modular in the interconnection scheme for multi-Interpreter systems as the number of Interpreters, memories or devices or path width changes.

These three characteristics provide the added benefit of decreasing design time by eliminating the need for new hardware developments for each new product.

THE INTERPRETER

Figure 1 is a summary diagram of an Interpreter. The five functional parts are tabulated below.

MCU	Memory Control Unit	Registers for memory addressing; expandable.
CU	Control Unit	Registers for conditional control and logic commands.
LU	Logic Unit	Width 1 to 8 bytes; data registers, adder, shifter.
MPM	MicroProgram Memory	Microprogram sequences: some words have literals, others have nano addresses.
NM	Nano Memory	Specific controls for the first three units.

This paper describes the results of effort supported, in part, by the Avionics Laboratory, Wright-Patterson AFB, Ohio, under Contract No. F33615-69-C-1200 and Contract No. F33615-70-C-1773.

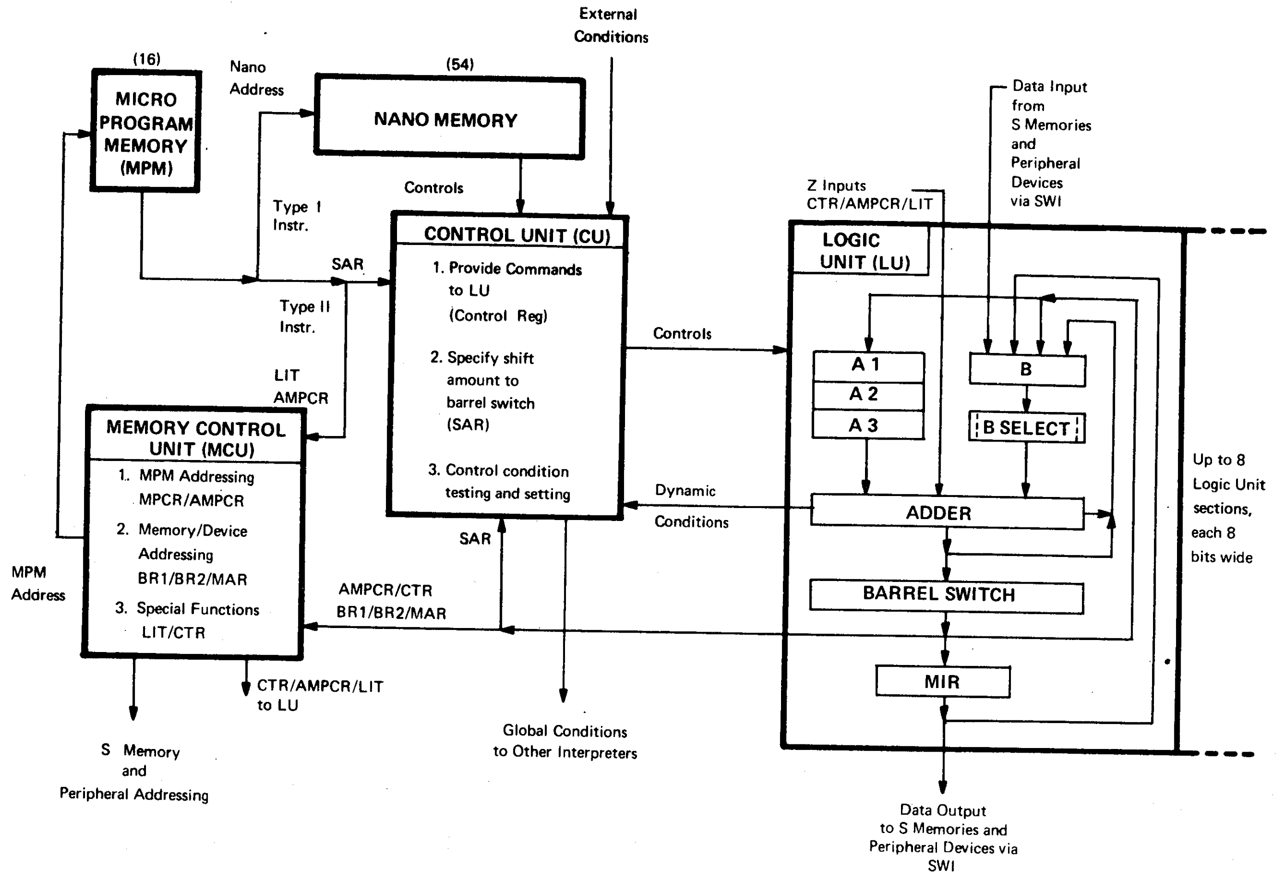


Figure 1. Interpreter Block Diagram

During the operation of the Interpreter, microprogram instructions and literals (data, jump addresses, shift amounts) are read out of the MPM. Data and jump addresses from the MPM are sent to the MCU, shift amounts, to the CU, and instructions are used as addresses for the NM. The output of the NM is a set of 56 controls which are transmitted to the CU, MCU and LU. The addressing of the proper locations in the MPM is handled by the selection of one of two microprogram count registers in the MCU and sending the contents of the selected register plus zero, one or two as an address to the MPM.

The LU performs the shifting and the arithmetic and logic functions required, as well as providing a set of scratch pad registers and the data interfaces to and from the Switch Interlock (SWI). Of primary importance is the modularity of the LU, providing expansion of the word length in 8-bit increments from 8 bits through 64 bits using the same functional unit. The CU contains a condition register, logic for testing the conditions, a shift amount register for controlling shift operations in the LU, and part of the control register used for storage of some of the control signals to be sent to the LU. The MCU provides addressing logic to the Switch Interlock for data accesses, controls for the selection of micro instructions, literal storage, and counter operation. This unit is also expandable when larger addressing capability is required.

Logic Unit (LU).

A functional block diagram of the LU is shown in Figure 1. The design of the LU is predicated upon implementation with one LSI silicon slice per 8 bits.

All A registers are functionally identical. They temporarily store data within the Interpreter and serve as a primary input to the adder. Any of the A registers can be loaded with the output of the barrel switch in one clock time. Selection gates permit the contents of any A register to be used as one of the inputs to the adder.

The B register is a primary external interface (from the Switch Interlock). It also serves as the second input to the adder, and can collect certain side effects of arithmetic operations. The B register may be loaded with any of the following (one per instruction):

1. The barrel switch output
2. The adder output
3. The data from the Switch Interlock
4. The MIR output
5. The carry complements of 4- or 8-bit groups with selected zeros (for use in decimal arithmetic or character processing)
6. The barrel-switch output ORed with 2, 3, or 4 above.

The output of the B register has true/complement selection gates which are controlled in three separate sections: the most significant bit, the least significant bit, and all the remaining central bits. Each of these parts is controlled independently, and may be either all ZEROs, all ONEs, the true contents or the complement (ONEs complement) of the contents of the respective bits of the B register.

The MIR buffers information being written to S memory or sent to a device. It is loaded from the barrel switch output and its output is sent to the Switch Interlock, or to the B register.

The adder in the LU is a modified version of a straightforward carry lookahead adder such as that discussed by MacSorley [11] and others. Therefore the details of its operation will not be included.

Inputs to the adder are from selection gates which allow various combinations of the A, B, and Z inputs. The A input is from the A register output selection gates and the B input is from the B register true and complement selection gates. The Z input is an external input to the LU and can be:

1. The output of the counter in the MCU into the most significant 8 bits with all other bits being ZEROs.
2. The output of the literal register in the MCU into the least significant 8 bits with all other bits being ZEROs.
3. An optional input (depending upon the word length) into the middle bytes (which only exists in Interpreters that have word lengths ⩾ 24 bits) with the most and least significant bytes being ZEROs. Usually this input will be the AMPCR.
4. All ZEROs.

Using various combinations of inputs to the selection gates, any two of the three inputs can be added together, or can be added together with an additional ONE added to the least significant bit. Also, all binary Boolean operations between any two adder inputs can be done.

The barrel switch is a matrix of gates that shifts a parallel input data word any number of places to the left or right, either end-off or end-around, in one clock time.

The output of the barrel switch is sent to:

1. A register
2. B register
3. Memory Information Register (MIR)
4. Least significant 16 bits to MCU
5. Least significant 3 to 6 bits to CU (depending on word length)

Control Unit (CU).

One CU is required for each Interpreter. The design of the CU is predicated upon implementation with one LSI silicon slice. This unit has five major sections: the shift amount register (SAR), the condition register (COND), part of the control register (CR), the MPM content decoding, and the clock control (Figure 2).

The functions of the SAR and its associated logic are:

1. To load shift amounts into the SAR to be used in the shifting operations.

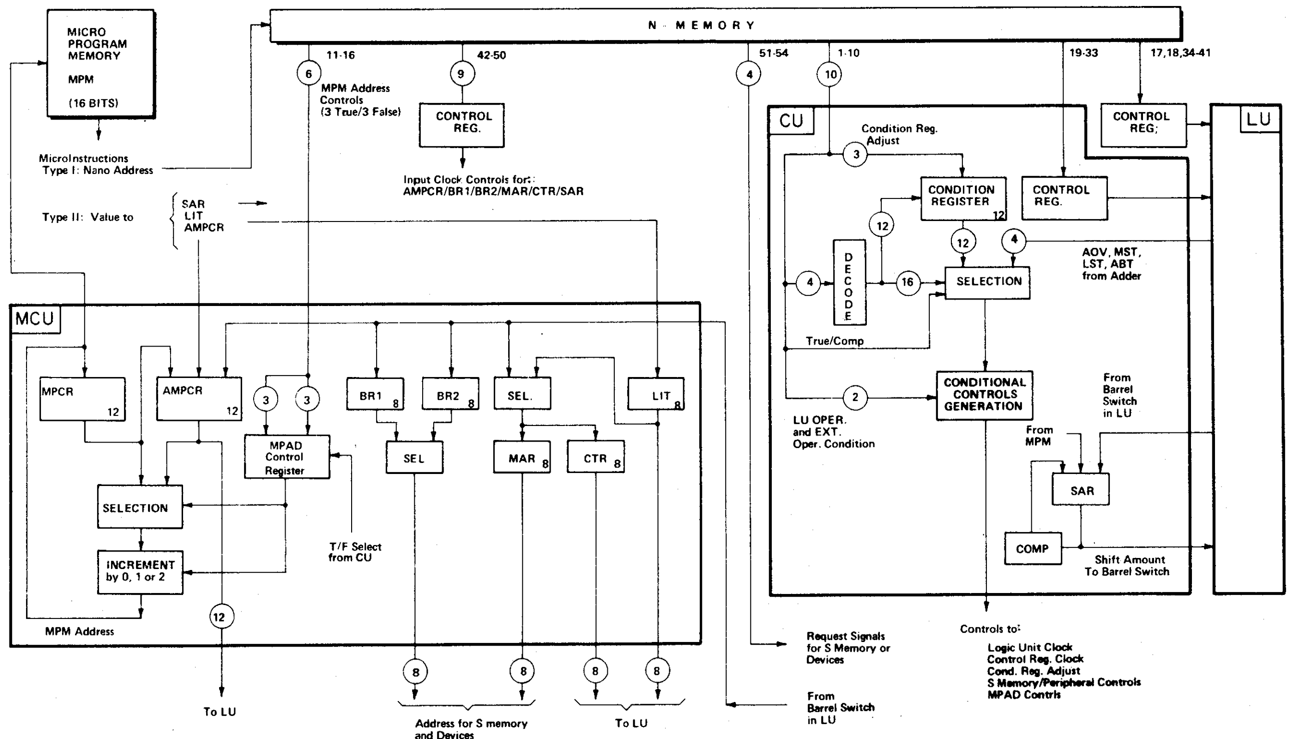


Figure 2. Interpreter Data and Control Flow

2. To generate the required controls for the barrel switch to perform the shift operation indicated by the controls from the Nanomemory.
3. To generate the "complement" of the SAR contents, where the "complement" is defined as the amount that will restore the bits of a word to their original position after an end-around shift of N followed by an end-around of the "complement" of N.

The condition register section of the CU performs four major functions:

1. Stores 12 resettable condition bits in the condition register.
2. Selects 1 of 16 condition bits (12 from the register and 4 generated during the present clock time in the Logic Unit) for use in performing conditional operations.
3. Decodes bits from the memory for resetting, setting or requesting the setting of certain bits in the condition register.
4. Resolves priority between Interpreters in the setting of global condition (GC) bits.

The 12 bits of the condition register are used as error indicators, interrupts, status indicators, and lockout indicators.

The control register is a 36-bit register that stores all control signals from the Nanomemory that are used in the LU, CU, and MCU for controlling the execution phase of a microinstruction.

The MPM content decoding determines the use of the MPM output (literal or address) based upon the first four bits of the MPM. Several decoding options are available.

Memory Control Unit (MCU).

One MCU is required for an Interpreter, but a second MCU may be added to provide additional memory addressing capability. The design of the MCU is predicated upon implementation with one LSI silicon slice. This unit has three major sections (shown in Figure 2):

1. The microprogram address section contains the micoprogram count register, the alternate microprogram count register, the incrementer, the microprogram address controls register, and their associated control logic. This section is used to address the MPM for the sequencing of the microinstructions.
2. The memory/device address section contains the memory address register, base registers one and two, the output selection gates, and the associated control logic.
3. The Z register section contains registers which are the Z inputs to the LU adder: a loadable counter, the literal register, selection gates for the input to the memory address register and the loadable counter and their associated control logic.

Nanomemory (NM).

The Interpreter is controlled by the output of the 56-bit wide Nanomemory which may be implemented with a read/write memory, a read-only memory, wired logic or a combination of the three. In any case, it has a typical form factor of 64 words by 56 bits, expandable in 64-word increments.

Each of the 56 bits represents a unique enable line to control the gates and flip-flops within the LU, CU, and MCU. Each Nanomemory word represents a specific microinstruction that is executed by the simultaneous presentation of a specific enable pattern for the 56 outputs represented by corresponding ONEs and ZEROs in its word.

Theoretically, a Nanomemory could provide 2^{56} (10^{17}) different word or output combinations. This number of words in a Nanomemory is obviously impractical. Somewhat fewer instructions would be considered reasonable. In fact, a set of 512 words has been found to be more than adequate for most Interpreter characterizations.

A unique feature of the Interpreter-Based System with its separate Nanomemory and Microprogram Memory is that the explicit enable lines for each microinstruction need be stored in the N memory only once, regardless of the number of times that a specific microinstruction is needed in a program. To accomplish this saving in memory, the Microprogram Memory contains not the full microinstruction needed, but rather the address in the Nanomemory where the explicit ONEs and ZEROs are stored that are needed to execute that instruction type. Thus, several microprogram sequences, each using the same microinstruction (e.g., transfer A to B) need only store in the Microprogram Memory the address of the Nanomemory word containing that operation.

Microprogram Memory (MPM)

Each Interpreter requires a source of microprogram instructions to define the operation of the Interpreter. To maintain the clock period of the Interpreter, this source must have a fast read access time and a cycle time less than the clock period just as for the N memory. Slower read access time memories may be used, but this will add directly to the clock period.

Two possible solutions for providing this source of microprogram instructions are listed below:

1. A semiconductor MPM. This memory can be a read-only memory (ROM) if the Interpreter is to be dedicated to the function defined by the ROM. A read-write memory can be used for experimental purposes or when the function of the Interpreter might be changed, such as reconfiguration in a multiple Interpreter system.
2. A buffer into a slower speed, wider word memory.

The advantage of splitting what is normally considered to be the microprogram memory into two parts is more graphically illustrated by comparing the total memory requirements of the two approaches shown in Figure 3. The total number of bits (T_1) in Figure 3(a) is $T_1 = 56M$. The total number of bits (T_2) in Figure 3(b) is $T_2 = Mk + 56N$. Figure 4 shows a plot of the total number of bits vs. M and N for both approaches. From this figure it is obvious that in some cases one memory is the proper approach. It should be remembered that the approach with two memories will require that the total access time through both memories be equal to the one memory approach resulting in memories that are more expensive per bit. Also of importance is that

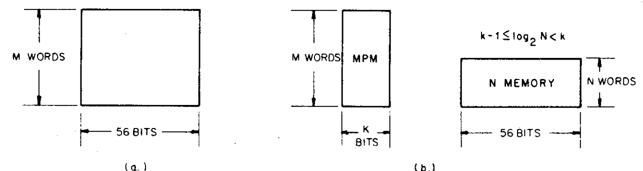


Figure 3. Two Methods for Implementing Microprogram Source

many applications require a writeable MPM. If the N memory is read only or decoding, a savings in cost results in the split memory approach due to fewer total bits that must be writeable.

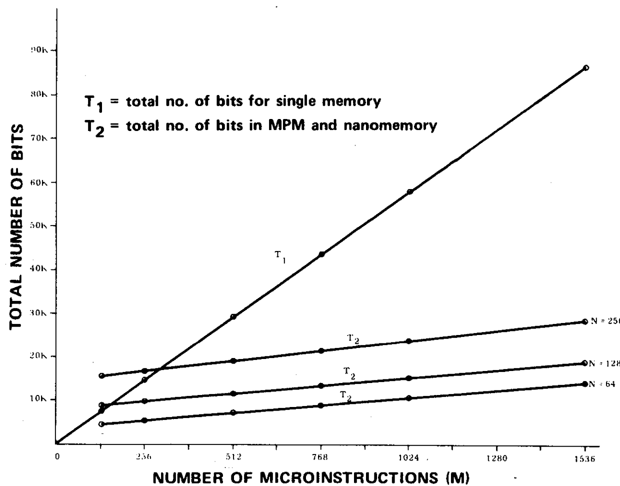


Figure 4. Plot of Total Size Versus Implementation Technique

MICRO PROGRAM TIMING AND SEQUENCING

A commercial user of an Interpreter Based System would write a program in an H-language (e.g., ALGOL) which would then be compiled into an S-language program, which would then be stored in an S-memory for execution. Each S-instruction is executed through the use of an M-program.

The M-program provides basically two functions:

1. Fetching and interpretation of the S-instruction. The interpretation depends primarily on the format(s) of the S-instruction.
2. Execution of the indicated S-instruction operation as defined by a set of M-instructions for that operation.

The N-instructions are in turn addressed by M-instructions.

A microinstruction may contain either a constant (Type II microinstruction) or the address of a nanoinstruction (Type I microinstruction). The function of the 56 enables in a nanoinstruction are summarized in Table 1. A Type II microinstruction may load the literal register, the shift amount register, or the alternate microprogram count register.

The execution of a microinstruction requires one or more sequential time periods, called phase 1, phase 2 and phase 3. The constant interval of time from the end of one clock pulse to the end of the next is a phase. Phases of successive microinstructions usually overlap, so that phase 1 of a current microinstruction is being executed while phase 2 or 3 of a prior microinstruction is also being executed.

For a type II microinstruction, phase 1 provides sufficient time to execute the instruction and no additional phases are required. For a type I microinstruction, the events taking place in each of the three phases are identified below. The timing relationship between Type I and Type II microinstructions is shown in Figure 5.

Table I. Nanocodes

Nano-Bits	Function
1-4	Select conditions
5	Selects true or complement of condition.
6	Specifies conditional or unconditional LU operation.
7	Specifies conditional or unconditional external operation (memory or DDP).
8-10	Specify set/reset of condition.
11-16	Microprogram address controls (wait, skip, step, etc.).
17-26	Selects A, B, and Z.
27	Carry control
28-31	Select Boolean and basic arithmetic operations.
32-33	Select shift operation.
34-36	Select inputs to A registers.
37-40	Select inputs to B register.
41	Enables input to MIR.
42	Enables input to AMPCR.
43-48	Enable and select input to address registers and counter (MAR, BR1, BR2, CTR).
49-50	Select SAR preset.
51-54	Select external operations (read, write, lock, etc.).
55-56	Not assigned.

1. Phase 1 always overlaps phase 2 or 3 of a prior Type I instruction. Phase 1 includes condition testing and adjusting, selection of the controls for next instruction address computation and initiation of S memory and device operations (conditionally) and Logic Unit operations (conditionally).
2. Phase 2 may require a variable number of clock times. It is used to delay phase 3 for conditions as shown in Figure 5.

   When fetching of the next instruction from the Microprogram Memory is delayed, the Type I instruction currently being executed (there always is one) remains in phase 2 until the fetch can be made. The duration of phase 2 is determined dynamically by the microprogram.

   When the next instruction is Type II, the current Type I instruction is held in phase 2 for one clock time, while the Type II instruction is executed.
3. Phase 3 requires one clock time. It always overlaps phase 1 of the next Type I instruction (there may be intervening Type II instructions). Phase 3 is used to perform the logic unit operations including the destination selection.

The actual sequencing of instruction fetches may best be understood by reference to Figure 2.

Words in the N memory are addressed by the output of the M memory. The M memory is addressed from the combination of the microprogram count register (MPCR), the alternate microprogram count register (AMPCR), and the 0/1/2 incrementer (INCR). The selection of the appropriate combination of data paths among these three is controlled by the microprogram address (MPAD) controls. These controls were selected as one of two alternative sets of controls from the N memory during the previous clock time, based upon the value of a condition selected during the same previous clock time. In the case of a Type II instruction, a fixed

A. Type I followed by Type I for which a logic operation is required.

TYPE I	ϕ 1	ϕ 3	
TYPE I		ϕ 1	

B. Type I followed by Type II, followed by Type I for which a logic operation is required.

TYPE I	ϕ 1	ϕ 2	ϕ 3
TYPE II		I I	
TYPE I			ϕ 1

C. Type I followed by Type I for which no logic operation is required, followed by Type I for which a logic operation is required.

TYPE I	ϕ 1	ϕ 2	ϕ 3
TYPE I		ϕ 1	
TYPE I			ϕ 1

Figure 5. Phasing of Type I and Type II Instructions

combination of controls (a STEP) will be forced into the MPAD control register independently of conditions, and the instruction currently being executed by the Logic Unit will be held in suspension (i.e., no results will be loaded into the destination registers) and the same operation will be repeated the next clock time as shown in Figure 5. This allows the next Type I instruction to make decisions based upon conditions being generated during the previous Logic Unit operation. It should be noted that the Type II instruction can affect the Type I instruction being held in suspension. For example, if the Type II loads the SAR, the Barrel Switch (BSW) will shift by the new contents of the SAR.

The successor microprogram address choices available are listed in Table II. The MPCR usually contains the address of the instruction currently being executed and the AMPCR usually contains the address of an alternative instruction minus one.

MICROPROGRAMMING EXAMPLES

A sample M-instruction is given by

A1 + B → A1, LMAR, CSAR;

This statement is one nanoinstruction that performs the following functions. The contents of register A1 are added to the contents of register B and the result is stored in register A1. The contents of the LIT (Literal register) are placed in the MAR (memory address register). The value of the SAR (shift amount register) is complemented. The capability is also provided for conditional branching to a subroutine or other instruction string with one of the eight successors listed above. In the above example, where no successor word is specified, the STEP instruction is implied, which means that the Interpreter will step to the next M instruction in sequence in the MPM. The above example could be made conditional as follows:

If LC1 then A1 + B → A1, LMAR, CSAR; jump else step;

This means if the local condition bit is equal to one, then do the indicated operations that follow after the THEN statement and JUMP to the instruction address that is currently in the AMPCR plus one. If LC1 is false then the action indicated after the ELSE statement takes place, i.e., STEP.

An example of a microprogram using M-instructions is given below for a fixed-point, binary multiply. The resulting average execution time is (6.5 + 2.5N) clocks where N is the number of bits in the magnitude of the operands.

Assumptions

(1) Sign-magnitude number representation

(2) Multiplier in A3; multiplicand in B

(3) Double length product required with resulting most significant part, with sign, in B and least significant part in A3

1. A3 XOR B →; if LC1

2. B_{0TT} → A2; if MST then Set LC1

   Comment: Step 1 resets LC1. Steps 1 and conditional part of 2 check signs; if different, LC1 is set.

3. B_{000} → B, LCTR

   Comment: Steps 2 and 3 transfer multiplicand (0 sign) to A2 and clear B.

4. "N" → LIT; 1 → SAR

   Comment: Steps 3 and 4 load the counter with the number (N = magnitude length) to be used in terminating the multiply loop and load the shift amount register with 1.

5. A3 R → A3 ;Save

   Comment: Begins test of least bit of multiplier and sets up loop

6. LOOP: If not LST then B_{0TT} C → B skip else step

7. A2 + B_{0TT} C → B

8. A3 OR B_{T00} R → A3, INC; If not COV then jump else step

   Comment: 6 thru 8 – inner loop of multiply (average 2.5 clocks/bit)

9. If not LC1 then B_{0TT} B; skip else step

10. B_{1TT} → B

    Comment: If LC1 = 0, the signs were the same, hence force sign bit of result in B to be a 0.

Table II. Successor Microprogram Addresses

Successor	MPM Address (MPAD)		Next MPCR Value	Next AMPCR Value
WAIT	MPCR		MPAD	*
STEP	MPCR	+ 1	MPAD	*
SAVE	MPCR	+ 1	MPAD	MPCR
SKIP	MPCR	+ 2	MPAD	*
JUMP	AMPCR	+ 1	MPAD	*
EXEC	AMPCR	+ 1	*	*
CALL	AMPCR	+ 1	MPAD	MPCR
RETN	AMPCR	+ 2	MPAD	*

*no change

MICROPROGRAMMING DESIGN TOOLS

There are two primary design tools for the microprogrammer. One is a microprogram translator (TRANSLANG) which is a computer program used to convert English language type statements defining the action of the Interpreter on each micro-instruction, into binary patterns for the M and N memories. TRANSLANG features include:

Error checking of TRANSLANG and hardware syntax

Collection of statistics on the static multiple use of N instructions

Lists program and contents of MPM and N-memory

Merging of TRANSLANG programs.

The second tool is an interactive clock by clock execution model that is driven by microprograms prepared using TRANSLANG. The model is written in APL. Model features include:

Variable byte width logic units, S-memory and devices

Event-time oriented interaction with S-memory or devices

Optional execution traces

Traps for state monitoring at execution of selected M or N words

Execution timing and dynamic event counts.

SWITCH INTERLOCK

The Switch Interlock (SWI) connects Interpreters with devices and S-memories. Connection with a device is by reservation for exclusive use by an Interpreter and is maintained until released. Connection with an S-memory module is for the duration of a single data word exchange, but is maintained until some other module is requested or some other Interpreter requests that memory module.

The SWI is intended to connect many Interpreters to many devices and to many memory modules. It is important to keep the number of wires in the crosspoints to a minimum. Consequently, a variety of combinations of serial and parallel data transmission paths are allowable. The amount of parallelism depends on the bandwidth necessary to complete the transmission in the number of clocks (usually one) required by the system. The serial transmission rate is significantly higher than the Interpreter's clock, and hence many bits can be transferred in one clock time.

Consistent with the building block philosophy of the Interpreter-Based System, the Switch Interlock is partitioned to permit modular expansion for increments of data and address path widths. Each incremental module is predicated upon implementation with LSI, as are the functional units of the Interpreter.

The six basic modules of the switch interlock are described below:

1. Memory/Device Controls (MDC) - This unit is used as an interface between the Interpreter and the control in (2) and (3) below, and as the control for the high frequency clock used for the serial transmission of data. There is one MDC per Interpreter.
2. Memory Controls (MC) - This unit is for resolving conflicts between Interpreters requesting the use of the same S-memory module and for maintaining an established connection after completion of the operation until some other module is requested or some other Interpreter requests that memory module. This unit handles two to four Interpreters and up to 8 S-memory modules. System expansion using this module may be in number of Interpreters or in number of memory module ports.
3. Device Controls (DC) - This unit resolves conflicts between Interpreters trying to lock to a device and checks the lock status of any Interpreter attempting a device operation. This unit handles up to 4 Interpreters and up to 8 ports. System expansion using this module may be in number of Interpreters or in number of device ports.
4. Output Switch Network (OSN) - This unit sends data from an Interpreter to an addressed device or data and address from an Interpreter to an addressed memory module, (i.e., the OSN is a "demultiplexer"). This unit handles two bits for up to 4 Interpreters and 8 device ports or memory modules.
5. Input Switch Network (ISN) - This unit returns data from an addressed device or memory module to the Interpreter (i.e., the ISN is a "multiplexer"). This unit also handles two bits for up to 4 Interpreters and up to 8 device ports or memory modules.
6. Shift Register (SR) - These units are optional and are parallel-to-serial shift registers or serial-to-parallel shift registers using a high frequency clock. These are used for serial transmission of data through the ISN's and OSN's.

Figure 6 is a block diagram of a Switch Interlock connecting up to 4 Interpreters to 8 device ports and 8 memory ports. The shift registers shown are optional and may be eliminated with the resulting increase in the width of the ISN and OSN transfer paths. Although it is not indicated by this figure, the switch interlock is expandable in terms of number of Interpreters, devices, memory modules, and path widths.

Overall Switch Interlock Control

The Interpreter has control over the SWI. Specifically, in an Interpreter based system utilizing one or more Interpreters, only Interpreters can issue control signals to access memories or devices.

A memory or device cannot initiate a path through the SWI. They may, however, provide a signal to the Interpreter via a display register or other similar external request device and may send control signals to the MDC. Transfers between devices and memories must be via and under the control of an Interpreter.

Controls are routed from the Interpreters via the MDC to the MC and the DC which, in turn, check availability, determine priority and perform the other functions that are characteristic of the switch interlock.

Switch Interlock Timing

There are no hardware timeouts in the SWI. Events are initiated by the Interpreter for access to memories or devices. The Interpreter awaits return signals from the MDC. Upon reception of these signals, it proceeds with its program. Lacking such positive return signals, it will either wait, or retry continuously, depending upon the Interpreter program (and not the SWI). Any timeout waiting for a response may be performed by either the programmer or a device that will force a STEP in the microprogram after a preset length of time.

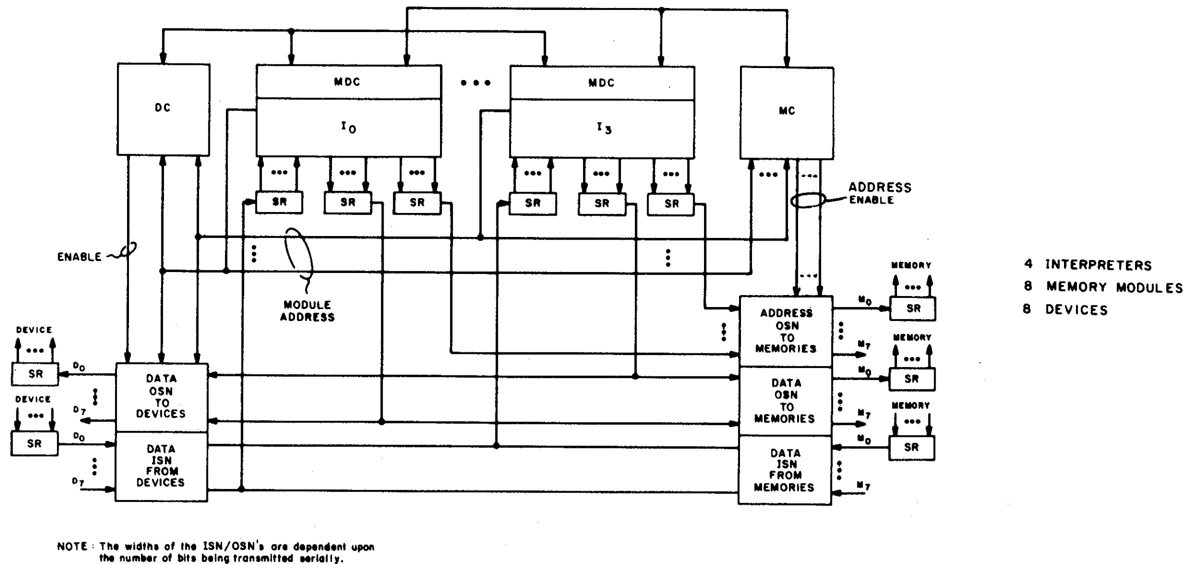


Figure 6. Switch Interlock

Among the significant signals which are meaningful responses to an Interpreter and testable as conditions are the following:

Switch Interlock has Accepted Information	The MAR and MIR may be reloaded and a memory or device has been connected.
Read Complete	Data is available to be gated into the B register.
EXT Request	Inspect for interrupt from device or memory.

The rationale for this approach is consistent with the overall Interpreter-based system design which permits the maximum latitude in the selection of memory, devices and their speeds. Thus the microprogrammer has the ability (as well as the responsibility) to provide the timing constraints for any system configuration.

Device Operations

The philosophy of device operations is based upon an Interpreter using a device for a "long" period of time without interruption. This is accomplished by "locking" an Interpreter to a device. The ground-rules for device operations are listed below:

1. An Interpreter must be locked to a device port to which a read or a write is issued.
2. An Interpreter may be locked to several device ports at the same time.
3. A device port can only be locked to one Interpreter at a time.
4. Since only the Interpreter that is locked to a device port can unlock it, when an Interpreter is finished using a device port it should be unlocked so other Interpreters can use it. The exception is the case where devices locked to a failed Interpreter may be unlocked with "privileged" instruction by another operative Interpreter.

Memory Operations

Memory-like modules normally cannot be locked and are assumed to require minimum access time and a short "hold" time of a memory module by any single Interpreter. Conflicts in access to the same module are resolved in favor of the Interpreter that last accessed the module, otherwise the highest priority requesting Interpreter. Once access is granted, it continues until that memory operation is complete. When one access is complete, the highest priority request is honored from those Interpreters then in contention. The Interpreter completing access is not able to compete again for one clock. Thus the two highest priority Interpreters are assured of access. Lower priority Interpreters may have their access rate significantly curtailed.

CIRCUITS AND PACKAGING

The circuit form chosen for the Interpreter is TTL. Tradeoffs for various logic forms are covered elsewhere [12].

TTL was chosen for the following reasons:

1. TTL has been used in military applications because it can be operated over large temperature excursions and has proven reliable.
2. Most bipolar LSI manufacturers are using TTL for their most complex array, which increases the confidence that LSI can be manufactured at a low cost.
3. The power dissipation of this circuit is such that the LSI wafer can be used without any significant cooling problems.

Implementations using two different manufacturing techniques are being pursued. One is using 14, 16 and 24 pin packages that are commercially available. The other is using discretionary wired LSI for the Air Force where volume is a primary concern. 50 – 80 gate MSI is being pursued as possible back-ups for both methods.

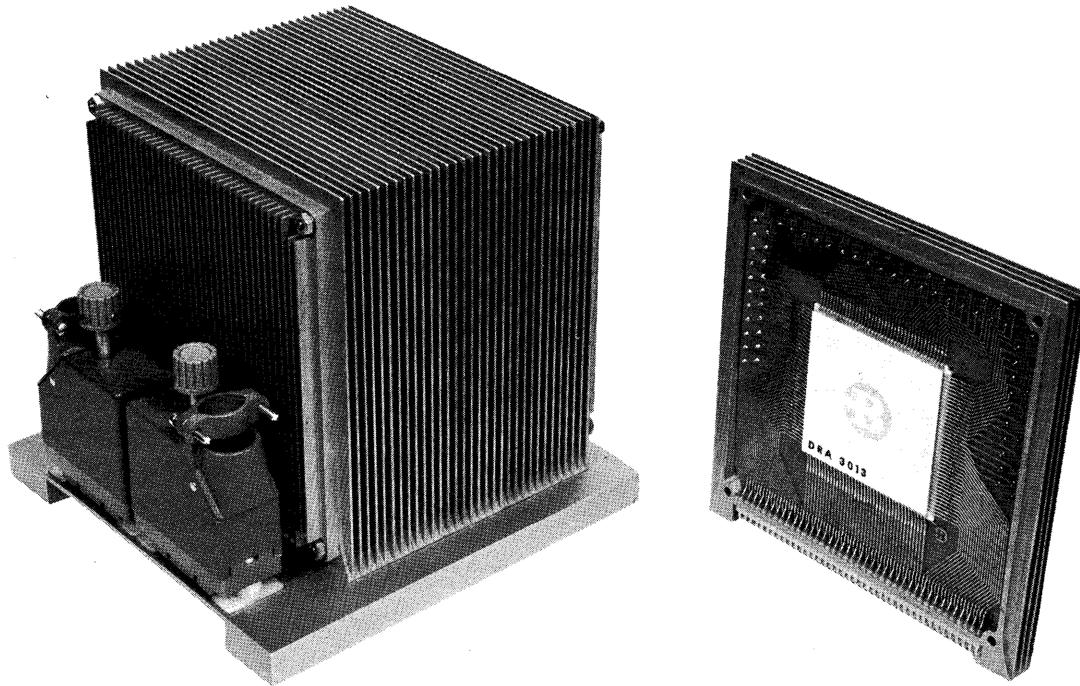


Figure 7. Interpreter

The LSI avionics multiprocessing system being built for the Air Force is scheduled for delivery in late 1971. It will consist of five, 32-bit Interpreters which are connected to each other, to four memory modules, and to up to eight device ports by a Switch Interlock. The LU, CU, and MCU functional modules of the Interpreters are each implemented with two discretionary wired LSI arrays mounted on opposite sides of a finned 5" X 5" X 1/2" (Figure 7). The MPM and N-memory are implemented with standard available flat packs and are mounted on the same castings. These are then sandwiched together for each Interpreter as shown in Figure 7. The Switch Interlock is implemented with a combination of custom MSI and standard complex function TTL, also partitioned to fit onto the same castings.

Each of the four main memory modules will initially be 4K X 36 bit core memories, being replaced by 16K X 34 bit plated wire memory modules for latter versions where power, weight, and volume require it. The entire multiprocessor with five Interpreters, the switch interlock and four 16K X 34 bit plated wire memory modules will occupy 1.8 cubic feet, weight 140 pounds and consume 570 watts.

INTERPRETER APPLICATIONS

Device controller design is an important microprogramming task. The objective is to provide as many logic functions as possible for device control through microprogramming. The only justifications for any logic other than level converters in the device interface is either the microprogram is simplified (saving MPM locations) or the Interpreter must be free to do other tasks in order to meet its requirements for device handling. By this means design flexibility is maintained, optional features are easily incorporated, and many special hardware controllers (typically one or more per device) are replaced by common hardware specialized through microprograms. The opportunity for shared use of one Interpreter among several devices, dynamically being interleaved as needed, represents a significant potential for system simplification. Use of the Interpreter as a device controller is presently being pursued.

A second application area being pursued for the Interpreter is for implementing stand-alone processors for small accounting machine applications where a microprogrammable processor provides the flexibility of using several different mixes of low-cost peripheral devices.

Emulation of an existing processor and/or its I/O channels is another microprogramming task. The objective is to run programs prepared for the emulated machine. To do this, the emulated machine registers are mapped into S-memory and/or the actual registers of an Interpreter with more frequently used registers such as the program count register or base address register usually being resident in the Interpreter. The operation codes accessible to programs become S-instructions. Other S-instructions may be added for I/O commands, depending on how much of the I/O processing is absorbed by the Interpreter.

Aside from the obvious advantage of microprogramming to emulation speed, other factors which aid emulation are the variable width of the logic unit to match the word length of the emulated machine, the one-clock time shifting of the barrel switch for breaking apart instructions, conditional testing, and true/complement selection of sections of the B register. Emulations of the Burroughs D 825 and B 300, as well as other machines, are currently being done.

A fourth application area is the emulation of higher-level language processors. One task presently being pursued is the direct implementation of an APL processor. A second task

being undertaken is the more global problem of designing and implementing a meta-compiler in the format of an extensible S-language which in turn will be used to build compilers for other problem oriented and higher level languages (such as FORTRAN and COBOL). The extensible language is implemented on a virtual network of processors with a distributed operating system.

A fifth microprogramming task is the development of an "S" language to aid in the construction of multiprocessing operating systems. Embedded in this language will be those operating system structures and functions used commonly and frequently by most control processes. This task is explored in Part II of this paper.

PART II. S MACHINE INTERPRETER LANGUAGE EMULATION

Multiprocessing is accomplished in our system by a combination of multiprogramming (interleaved execution) and parallel processing (simultaneously running multiple processors or interpreters). Microprogramming allows us the opportunity to develop a particular set of system characteristics using a very basic but general collection of hardware tools. Burroughs has been involved in the concept of integrating hardware and software as a design principle for many years [13]. A microprogrammed interpreter now allows for a "soft" or "virtual" computer which enables a system designer to develop a unique instruction set ("S" language) for his particular system.

Multiprogramming requires the ability to interleave the execution of processes. Thus an Interpreter is not exclusively allocated to a process for its entire execution time but may be shared by many processes. The system accomplishes multiprogramming by the technique of queueing and dynamic resource allocation. Each process is a set of resources which must contain all of the information necessary to describe its own status during execution as well as during the waiting periods for an Interpreter. This concept is implemented by creating for each process its own unique work area containing a stack. Thus two resources always necessary to every process in order to function are a unique work area and an Interpreter.

Parallel Processing occurs when more than one Interpreter is available on a system. Processes may then be executed simultaneously. Since the system structure is such that an Interpreter is treated as another resource, Interpreters may be easily added or deleted from the system. One additional mechanism needed for parallel processing is the "Lock" instruction. This will prevent simultaneous accessing of data or execution of code by independent Interpreters (e.g., two Interpreters simultaneously allocating the same free memory space). An Interpreter entering system tables must lock out all other Interpreters until it becomes safe for them to proceed again without the danger of conflicts.

The primary objective of an operating system is to optimize and synchronize the process to be run and to allocate to them the available system resources [14]. Operating systems are organized about tables, lists, queues of stacks, used for the purpose of storing of information concerning the resources available as well as the processes needed to run. Communication between modules of the system are accomplished through queues. Thus this system contains instructions which allow for the easy handling of tables and other data structures. The flexibility afforded an operating system in performing its tasks of resource allocation is related to the time at which the binding of programs and data takes place. In a multiprocessing system binding of resources must occur close to execution time since a single process must not

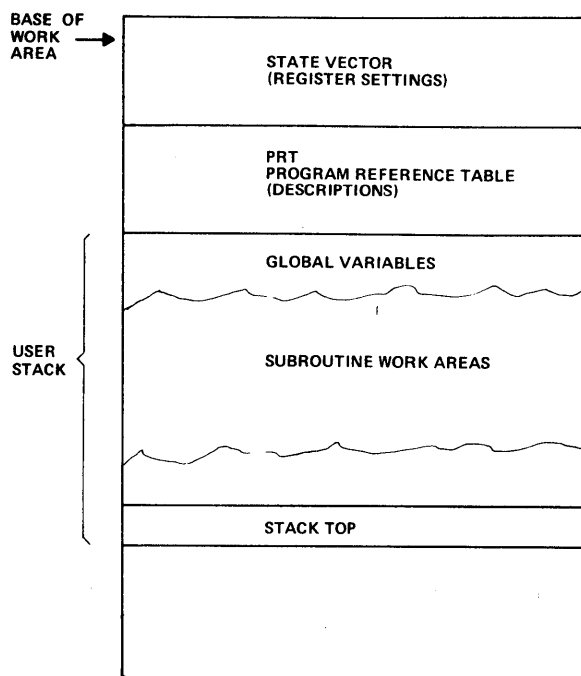


Figure 8. Work Area

be permitted to affect the binding policy of the system. Resources must also be released as soon as possible to permit reallocation to others. In our system we include base registers, and description driven resource allocation which aid in the deferment of binding.

The "S" language developed for the multiprocessing system will include part of the operating system in its control structure. Venus describes a similar undertaking [15]. Each instruction accessing data or program can detect the absence from memory of this data, and can directly retrieve it from secondary storage after allocating memory for it. The detection and handling of interrupts will also be built into the system language. However, many modules of the operating system will be written in the developed "S" language and will be just another process to be run. Included in these functions will be scheduling of processes, handling of external communications, initialization and termination of processes. The following is a description of the "S" language features of the system under development. The language is called "SMILE" which stands for "S" Machine Interpreter Language Emulation.

WORK AREA

Every process in the system must have its own unique work area (Figure 8). The work area of a process contains:

1. State Vector. The registers and temporary storage used by the "SMILE" firmware which fully describes its status.
2. Program Reference Table (PRT) [16]. A list of the descriptions of the program segments as well as the data and file segments reserved for this process.
3. User Stack. Provides the process with the facility for temporary storage of data and a dynamic history of the process.

The top of the stack is a quick access environment for data manipulation. The ability to transfer control to a remote subroutine and return is provided by the stack mechanism. The bookkeeping required to save the address of the calling routine and reserve working area for the subroutine is provided by the instruction logic defining subroutine transfer.

ADDRESSING

The program work area is placed in a contiguous block of memory starting anywhere in memory. The starting address of any process is its base work area register, or the address of the start of its work area. To initiate a process all that is needed is to set the base work area register in the Interpreter. Since the work space contains the state vector of the process all the information necessary for running is available. When a process references an object in its work area a relative address is used. Thus absolute addresses are not needed at run time.

When an instruction references through the PRT part of the work area, it accesses a description found there. Descriptions are the only objects which may contain absolute addresses for locating program segments or data.

DESCRIPTION

All resources, program or data in the SMILE machine are described by one or more descriptions. Descriptions are viewed as programs whose evaluation produces the desired items [16]. They are words used to locate data and program and to describe these areas for control purposes. When executed, a description causes firmware to fetch, use or replace the desired object similar to the way an instruction performs a given function. By describing data with descriptions, a program becomes data independent and information is kept out of the program stream (i.e., an object format may be unknown and a program can still perform a specific function on a description-defined set of data.)

There are several fields to be evaluated in a description. The format fields define the structure and the format of the data. Length and location fields specify the location of the objects, the size of the object and any limits imposed. The qualification fields control access and govern the data usage. Operating system flags are also included in descriptions. There are two types of descriptions in the "S" machine. The first is an indirect pointer which is used mainly to define the format of the object to be accessed. It also may contain the qualification fields and always points to another description (Figure 9).

The other is the direct description which is a pointer to the desired object space. A direct description alone always refers to a machine word sized object.

Each bit in a description is evaluated and causes the firmware to perform a specific function.

	O S FLAGS	QUALIFICATION	FORMAT	LOC FIELD
Initial Indirect	P | M | D | T	C	ET | ST | UNIT	R | NR | ADDRESS
Holder Indirect	P | M | D | T	–	Holder	R | NR | ADDRESS
Direct	P | M | D | A | B	C	Size	R | G | ADDRESS

Figure 9. Descriptions

Operating System Flags

The presence of operating system flags causes an operating system function to be executed.

P (Presence Bit): When the presence bit is on it means that the object to be accessed is not necessarily in memory and must be retrieved using the firmware developed as the operating system allocate memory function. In the case of an indirect description the next object to be retrieved is another description.

M (Monitor): The object, when finally accessed, should be monitored by the operating system, if any one of the M bits along the access path is on. The monitor bit is used by the operating system for checking the use of special data.

D (Direct Description): When the direct description bit is on the description being processed is treated as a direct description or the end of an evaluation chain. Otherwise it is assumed that the present description is pointing to another description.

T (Type of indirect description): When the T = 1 the indirect description is an extension description and contains a holder field which defines a subfield containing the previously defined objects. If T = 0 the description defines the object desired, the control imposed and general structure of the space and element to be accessed.

AB (Alter Bit): This bit is used by the operating sytem functions during memory allocation. If on, it indicates that the object in memory has been altered during its residence in main memory. Thus an object must be copied back to secondary storage only if it has been altered.

Qualification Field

The qualification field determines how the defined object can be used.

C (Controls): The controls imposed upon the user may change according to the access path he is assigned. If both the direct and indirect description have different controls imposed, the user will abide by the most restrictive of the two controls during the access.

For C = 00 Read write allowed
01 Read append (write not allowed)
10 Read only
11 Execute only

Format Field

The format fields define the structure of holder elements, the nested structure of space and the byte or unit size of the smallest element. It may also define the type of element being accessed.

ET (Element Type): The object or objects selected are of type integer if ET = 0 or of type floating point if ET = 1.

ST (Structure): The structure of the most global space is defined in the initial description. All of the other spaces nested within this space are assumed to be vector spaces. If no initial description is used, then the space is assumed to be a vector space.

For ST = 00 Vector space
01 Stack structured space
10 Queue structured space
11 Link list structured space

U (Unit): The unit is defined as the size of the basic measure within a structure. The unit is calculated as:

For Unit =	
0	1 bit
1	2 bits
2	4 bits
3	8 bits
4	16 bits
5	32 bits
6	64 bits

Holder (Nested space container): Contains the number of Units or previously defined holder fields which can fit into this next substructure. If the previous description defined a Unit of 2, (4 bits) then a holder value of 10 will indicate 10 4-bit units can fit in a holder (or a 40 bit element). If the next holder description indicates 3 of the above holders can fit into the next sub-space, then it is defining a 120-bit sub structure.

Location Field

The location field indicates where the data may be found. This may be an absolute address in "S" memory or an address relative to the work area. A presence bit set may indicate that the object is not in main memory. Thus a different kind of address identification will be necessary to locate the data.

R (Relative): When this field is set, the address is assumed to be an address relative to the work area.

NR (Name Register): If NR = 0 then no address modification is executed when accessing the next description in the chain.

If a name register field is selected, the value in the UP field (described later) of the specified name register is used to modify the address field.

If a direct description evaluation is being performed then the starting address is formed by using the global name register (if G = 1) coupled with all the format information gathered during the description evaluation cycle (Figure 10).

G (Global): If G = 1 then the global name register will be used in the calculation of the address for accessing the desired object. If no indirect description has been used the unit length is assumed to be word size. If G = 0 no global name register is needed and the first or next (depending on the structure) object is accessed.

The length of fields must be located in the description. This will bound the operations setting the limiting factors on data fields.

(Size): The number of elements of the most global defined holder field which can fit into this defined space.

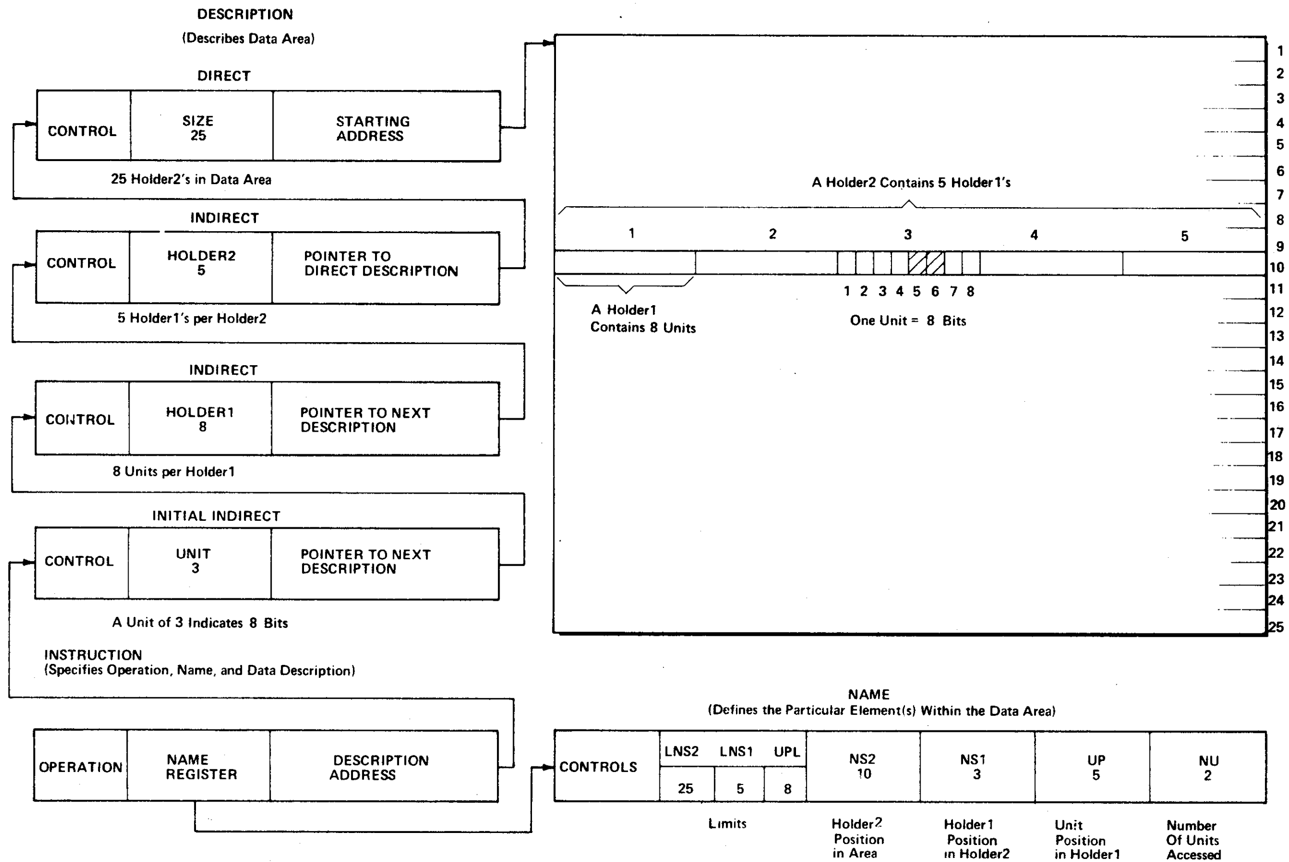


Figure 10. Data Accessing

NAME REGISTERS

The name registers allow for the naming of a particular element or group of elements in a structure during instruction execution. It may be used as a means of looping through a commonly named group of elements or just using a single element. The format of this register is described in Figure 11.

There are eight name registers availabe for use NR1 ....NR7 and GNR (the global name register). The global name register is the one used for developing the final name during description evaluation. When X1 = 1 then a nested space structure must be evaluated to locate the desired element. If X1 = 0 then there is a single set of elements within the structure to be evaluated. The X2 bit is only checked if X1 = 1. It is used to indicate if more than one level of nesting exists within the structure. When M = 1 then a multi element type operation will be performed by the instruction, (i.e., a search instead of a compare). If M = 0 then a single element operation will occur.

X1	X2	M	I0	I1	I2	–	NU	UPL	UP	LNS1	NS1	LNS2	NS2

0 1 2 3 4 5 6 8 16 24 32 40 48 56 64

Figure 11. Name Register

I0, I1 and I2, used only if M = 1, are the flags which specify which of the nested indices change during a multielement type operation (Figure 12(a) through 12(g)). The NU field defines the total number of units to be accessed during instruction execution. If NU = 0 or 1, one unit is used. The unit size of an operation is defined in the indirect description. If it is not defined then it is assumed to be word size.

The unit position (UP) is the position of the smallest unit within the next larger vector. If UP = 5 then it indicates the fifth unit within the containing space. The maximum UP allowed is found in Unit Position Limit (UPL). NS1 contains the position of the smallest vector structure within the next largest structure within the containing physical area. The maximum NS2 allowed is defined in LNS2. To address a bit position in memory within a structure the following formula is used:

BP = (((NS2-1 X Holder2) + NS1-1) Holder 1 +
UP-1) UNIT

Address = Starting Address + BP

DATA STRUCTURES

There are four built-in basic regular data structures defined for information written in the SMILE language: vector, stack, queue, and link list. All objects within a regular structure are of equal length. To access different field lengths the NU field of the name register is used to define the number of units in a given access.

INSTRUCTION SET

The length of a "SMILE" instruction will be mostly 8 bits or one byte size. However, there will be an escape bit for allowing 16 or 24 bit (2 and 3 byte) instructions. The one byte instructions use the stack top and/or the accumulator for accessing descriptions and/or variables. The longer instructions assume a relative address for accessing data.

The power behind each instruction is due to the description mechanism. When accessing a vector a simple compare becomes a search or an add may become a summation. Field isolation is done automatically so that data may be packed in the most efficient way conceived without being hampered by word boundaries. "SMILE" is not complete and instructions may be changed, added or deleted until an ideal set is found. The order code is totally soft as is the selected register set. Thus the language requirements may be modified until the designed system is firmly developed. Special functions may require special instructions. These instructions may be appended to "SMILE" for individual customer needs (i.e., a special microprogram for executing a matrix multiply as an instruction).

CONCLUSIONS

Microprogramming has proven its worth in many applications. The above-described data descriptions of the SMILE language are an example of this. To implement equivalent functions in software (as is done now) consumes considerable amounts of processing time. To implement equivalent functions in hardware would provide much faster execution times, but would result in hardware so complex as to be impractical. However, when implemented in firmware on a microprogrammable Interpreter, most of the advantages of hardware implementation are provided without the accompanying hardware complexity.

Up to this time, microprogrammed systems have suffered from one major limitation: because they were so simple (in which lies their chief virtue) they were rather limited in their processing throughput and were thus suitable only for relatively small, dedicated tasks.

What has been needed is a technique to interconnect many small micro-programmed computers into one system, and a capability to control this array of processors so that they can function efficiently while dynamically sharing the load.

The Interpreter-based system represents such a technique. The Interpreter itself is a modern, fourth generation microprogrammable computer suitable for LSI implementation. The Switch Interlock is the means for interconnecting virtually any desired number of Interpreters into a unified system with memories and peripheral devices. The SMILE language represents the firmware necessary to control and efficiently utilize these interconnected Interpreters. Not only does this technique allow the flexibility of micro-programming to be applied to large-scale systems, but it also allows systems of virtually any size to be constructed and provides for smooth evolution from a very small system to a very large one. Furthermore, it provides a degree of simplicity in logistics and maintenance not previously possible in medium or large scale systems, because the entire system is constructed using relatively few different types of simple modules. Thus, the Interpreter-based system appears ideally suited for Avionics processing of the 1970s, and for many other applications as well.

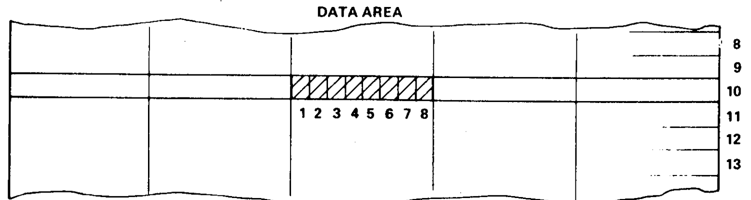
ACKNOWLEDGEMENTS

The authors would like to acknowledge the work of their collegues in Advanced Development at Burroughs on the development and use of the Interpreter, and the work of U. Faber on the development of the Switch Interlock. Special mention should go to J. T. Lynch, Director of Advanced Development and also to R. Conklin and D. Brewer of Wright-Patterson AFB for their support and encouragement.

NAME

CONTROLS	OTHER LIMITS	UPL 8	NS2 10	NS1 3	UP 1	NU 1

Access one unit at a time

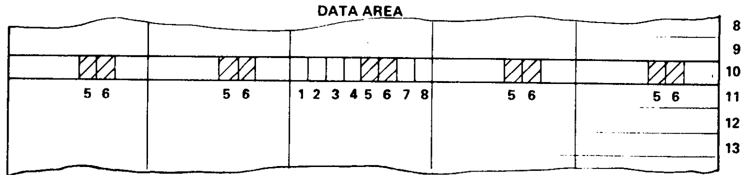


(a) Multioperations I0 = 1

NAME

CONTROLS	OTHER LIMITS	LNS1 5	NS2 10	NS1 1	UP 5	NU 2

Access two units at a time

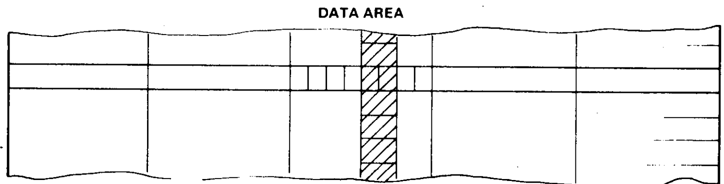


(b) Multioperations I1 = 1

NAME
I0 = 0 I1 = 0 I2 = I

CONTROLS	OTHER LIMITS	LNS2 25	NS2 1	NS1 3	UP 5	NU 2

Access two units at a time



All Holder 2 elements are accessed

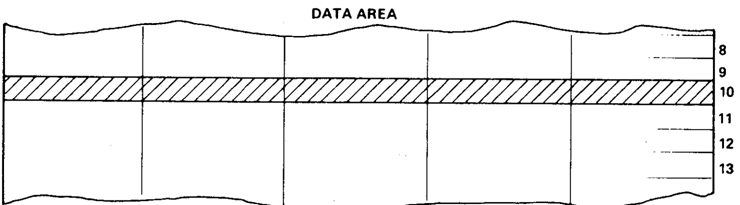
(c) Multioperations I2 = 1

NAME
I0 = 1 I1 = 1 I2 = 0

CONTROL	OTHER LIMITS	LNS1 5	UPL 8	NS2 10	NS1 1	UP 1	NU 1

Access one unit at a time

(The entire 10th Holder2 element will be sequenced through one unit at a time)



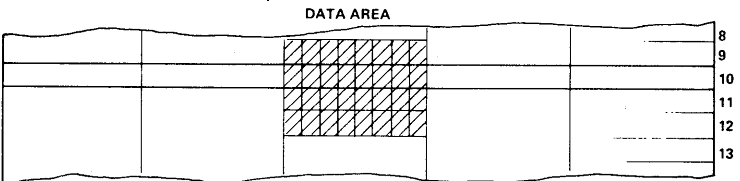
(d) Multioperations I0 and I1 = 1

NAME
I0 = I I1 = 0 I2 = I

CONTROL	OTHER LIMITS	LNS2 12	UPL 8	NS2 9	NS1 3	UP 1	NU 1

Access one unit at a time

(Each unit of the third Holder1 element in the 9th through 12th Holder2 element will be used)



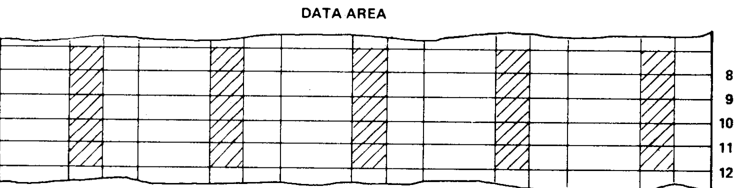
(e) Multioperations I0 and I2 = 1

NAME
I0 = 0 I1 = 1 I2 = 1

CONTROL	OTHER LIMITS	LNS2 12	LNS1 5	NS2 8	NS1 1	UP 5	NU 2

Access two units at a time

(Two units will be accessed each time the operation sequences through all 5 holder1's in the 8th through 12th holder2 element)



(f) Multioperations I1 and I2 = 1

Note: When I0, I1, and I2 are all ONE's, the operation sequences through all unit elements.

Figure 12. Multi Operation

REFERENCES

1. M.V. Wilkes, "The Growth of Interest in Microprogramming: A Literature Survey," *Computing Surveys* vol. 1, no. 3, pp. 139-145, September, 1969.

2. R.F. Rosin, "Contemporary Concepts of Microprogramming and Emulation," *Computing Surveys,* vol. 1, no. 4, pp. 197-212, December, 1969.

3. C.V. Ramamoorthy and M. Tsuchiya, "A Study of User-microprogrammable Computers," *AFIPS Conf. Proc.* (SJCC), vol. 36. Montvale, N.J.: AFIPS Press, pp. 165-181, 1970.

4. S.S. Husson, *Microprogramming: Principles and Practices,* Englewood Cliffs, N.J.: Prentice Hall, 1970.

5. A.J. Critchlow, "Generalized Multiprocessing and Multiprogramming Systems," *AFIPS Conf. Proc.* (FJCC), vol. 24, New York, N.Y.: Spartan Books, pp. 107-126, 1963.

6. B. Wald, "Throughput and Cost Effectiveness of Monoprogrammed, Multiprogrammed, and Multiprocessing Digital Computers," NRL Report 6549, Project No. RF-001-08041, April, 1967.

7. G.R. Blakeney, et. al., "An Application-oriented Multiprocessing System: Design Characteristics of the 9020 System," *IBM Systems Journal* vol. 6, no. 2, pp. 80-94, 1967.

8. B. W. Lampson, "A Scheduling Philosophy for Multiprocessing Systems," *Comm. ACM,* vol. 11, no. 5, pp. 347-360, May, 1968.

9. A. S. Buchman, "Aerospace Computers," *Advances in Computers,* vol. 9, F. L. Alt and M. Rubinoff, Eds. New York: Academic Press, 1968.

10. D.O. Baechler, "Aerospace Computer Characteristics and Design Trends," *Computer,* vol. 4, no. 1, pp. 46-57, January/February, 1971.

11. O. L. MacSorely, "High Speed Arithmetic in Binary Computers" *Proc. IRE,* pp. 67-91, January 1961.

12. L. .S. Garret, "Integrated-Circuit Digital Logic Families, *IEEE Spectrum,* vol. 7, no. 10, pp. 46-58, October 1970; no. 11, pp. 63-92, November, 1970; no. 12, pp. 30-42, December 1970.

13. "The Descriptor - A definition of the B 5500 Information Processing System," Burroughs Corp., February, 1961.

14. R.F. Rosin, "Supervisory and Monitor Systems," *Computing Surveys,* vol. 1, no. 1, pp. 37-54, March, 1969.

15. B.J. Huberman, "Principles of Operation of the Venus Microprogram," Mitre Technical Report 1843, May 1, 1970.

16. " A Narrative Description of the Burroughs B 5500," Burroughs Corp., May, 1965.

17. R. Barton, Private conversations, 1967-1969.